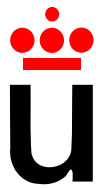
Concurrencia básica

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

22 de abril de 2022



Universidad
Rey Juan Carlos

Concurrencia

- Ya hemos visto varios problemas de concurrencia. P. ej.:
  - ¿Ejecuta antes el padre o el hijo?
  - ¿Qué pasa si dos procesos intentan escribir un fichero *a la vez*?
  - ¿Cómo se sincronizan dos procesos usando un pipe?
  - ...

ver fork0.c

Condición de carrera

- Recuerda: tenemos una **condición de carrera** cuando el resultado depende de qué flujo de ejecución llegue el primero y **no sabemos qué pasará**.
- El problema se complica si tenemos **memoria compartida**: varios flujos de ejecución leen/escriben variables compartidas concurrentemente.

Condición de carrera

- Una **condición de carrera** es el peor bug posible.
- En general, no es reproducible.
- Ocurre con cierta probabilidad (puede ser muy baja).
- Los resultados pueden ser sorprendentes y **muy difíciles de entender**.

Ejemplo 0

Ejemplo[1]: la variable x (inicializada a 0) está compartida entre dos flujos de control (A y B) que ejecutan este código, ¿cuáles son los posibles valores finales de la variable x? ¿Máximo y mínimo?

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++){
3:               aux = x;
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

[1] Suponiendo que las líneas de este programa son atómicas.

Ejemplo 0

(1) A ejecuta hasta la línea 3:

```
1:          int i = 0, aux = 0;        <= B(x=0,aux=0,i=0)
2:          for(i=0; i<10; i++){
3:               aux = x;              <= A(x=0,aux=0,i=0)
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

Ejemplo 0

(2) B ejecuta 9 iteraciones y se queda en la línea 2:

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++){   <= B(x=9,aux=9,i=9)
3:               aux = x;          <= A(x=9,aux=0,i=0)
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

Ejemplo 0

(3) A ejecuta su primera vuelta, se queda en la línea 2 y deja `x` a 1:

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++){  <= B(x=1,aux=9,i=9), A(x=1,aux=1,i=1)
3:               aux = x;
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

Ejemplo 0

(4) B avanza hasta la línea 3, su aux queda con valor 1:

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++){   <= A(x=1,aux=1,i=1)
3:               aux = x;          <= B(x=1,aux=1,i=9)
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

Ejemplo 0

(5) A ejecuta todas sus vueltas:

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++{
3:               aux = x;          <= B(x=9,aux=1,i=9)
4:               aux = aux + 1;
5:               x = aux;
6:          }
                                   <= A(x=9,aux=9,i=10)
```

Ejemplo 0

(6) B termina su última iteración:

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++){
3:               aux = x;
4:               aux = aux + 1;
5:               x = aux;
6:          }
                                  <= A(x=2,aux=9,i=10),  B(x=2,aux=2,i=10)
```

$$x = 2$$

Ejemplo 1

Dos flujos de ejecución A y B que comparten la variable `x` inicializada a 0 ejecutan estas sentencias concurrentemente:

```
x++;
printf("%d\n", x);
```

Posibles resultados:

- A imprime 1 y B imprime 2.
- B imprime 1 y A imprime 2.
- A y B imprimen 2.
- Dependiendo en las instrucciones que requiera el incremento en la arquitectura, pueden imprimirse cualquier par de enteros.

Atomicidad

- Si una operación sobre el recurso compartido no es **atómica**, dos flujos pueden interferir entre ellos $\rightarrow$ resultado de las operaciones incorrecto.
- **Operación atómica**: operación indivisible, ninguna otra operación que realice otro flujo puede interferir con ella.
- En general, pocas operaciones son atómicas y debemos sincronizar los flujos.

Ejemplo 2

En el mismo escenario que el Ejemplo 1, con una variable compartida adicional llamada `busy` que está inicializada a 0:

```
if(! busy){
      busy = 1;
      x++;
      printf("%d\n", x);
      busy = 0;
}
```

¿Hemos eliminado la condición de carrera? **No**:

- A evalúa la condición del if: true.
- B evalúa *"a la vez"* la condición del if: true.
- A ejecuta el cuerpo del if mientras que B ejecuta el cuerpo del if.
- Mismos posibles resultados que en el Ejemplo 1.

Exclusión mutua

- Lo que estaba intentando hacer el Ejemplo 2 sin éxito es proporcionar **exclusión mutua**: cuando un flujo está dentro de una región, no puede entrar ningún otro.
- Se llama **región crítica** a esa región del programa donde se debe proporcionar exclusión mutua para evitar una colisión al usar un recurso compartido.

Exclusión mutua

- Para hacerlo bien, necesito una operación que compruebe y escriba la variable `busy` de forma atómica.
- Inhabilitar interrupciones **no** es suficiente:
  - Cada CPU tiene sus propias interrupciones $\rightarrow$ ¿funciona en un multiprocesador?
  - No es deseable que los procesos de usuario puedan desactivar todas las interrupciones de la CPU $\rightarrow$ ¿qué pasa si hay un bug?
- ¡Necesito soporte hardware! (cerrar bus de memoria, evitar interrupciones, etc.)

TAS

- Los procesadores incluyen operaciones atómicas como test-and-set (TAS):
  - Si está a *false*, lo pone a *true* y devuelve *false*.
  - Si está a *true*, lo pone a *true* y devuelve *true*.
- Algunos procesadores no tienen TAS, pero sí tienen otras operaciones atómicas que nos permiten implementar TAS.

Instrucciones equivalentes a TAS

Por ejemplo, en AMD64 tenemos XCHGL:

- Intercambia los dos valores de forma atómica.
- Del manual de intel:

[...] the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL [...]

Implementación de TAS en Plan 9/AMD64

```
TEXT        _tas(SB),$0
        MOVL        $0xdeadead,AX  // escribe literal en AX
        MOVL        1+0(FP),BX     // escribe puntero en BX
        XCHGL       AX,(BX)        // intecambia AX con *BX
        RET                        // retorna AX
```

Primitiva: lock/cerrojo

- Con TAS ya podemos tener exclusión mutua...
- ... pero es deseable tener una abstracción de más alto nivel, que sea más fácil de usar que TAS.
- Con TAS podemos implementar la primitiva **lock (cierre o cerrojo)**.

Primitiva: lock/cerrojo

- Un cerrojo tiene dos operaciones:
  - lock(): coge el cierre.
  - unlock(): suelta el cierre.
- Para entrar en la región crítica (i.e. usar el recurso compartido), hay que tener cogido el cierre.
- Si un cierre está libre, se puede coger.
- Si se intenta coger un cierre ya cogido, el flujo se queda bloqueado hasta que pueda cogerlo.
- Después de usar el recurso compartido, siempre se debe soltar el cierre.

Ejemplo 3

En el mismo escenario que el Ejemplo 1, con una variable compartida adicional `lk` de tipo cerrojo:

```
lock(&lk);
x++;
printf("%d\n", x);
unlock(&lk);
```

¿Hemos eliminado la condición de carrera? Ahora el contador es coherente. Ya sólo pueden pasar dos cosas:

- A imprime 1 y B imprime 2.
- A imprime 2 y B imprime 1.

Si queremos un orden específico, necesitamos más sincronización.

Primitiva: lock/cerrojo

- ¿Y si quiero proteger distintos recursos compartidos? Para optimizar el uso concurrente, puedo usar distintos cerrojos...
- Problema cuando hay varios cerrojos: **deadlocks** o interbloqueos. P. ej.:
  - A coge lk1, espera a que B suelte lk2
  - B coge lk2, espera a que A suelte lk1
  - A espera a B, pero B no acaba porque espera a A $\rightarrow$ dependencia circular, **ninguno progresa**.

```
// A ejecuta...

lock(&lk1);
lock(&lk2);
//blabla
unlock(&lk2);
unlock(&lk1);
```

```
// B ejecuta...

lock(&lk2);
lock(&lk1);
//blabla
unlock(&lk1);
unlock(&ll2);
```

Primitiva: lock/cerrojo

Hay que tener mucho cuidado con los locks. En general:

- Coger los locks siempre en el mismo orden.
- `lock()`/ `unlock()` en la misma función.
- `lock()`/ `unlock()` al mismo nivel de tabulación.
- Todos los locks al mismo nivel de abstracción.
- Tener claro qué funciones cogen el lock y cuales no.

Buenas prácticas: cada return con su unlock

Mal:

```
lock(&lk);
if (bla) {
                    dosomething();
                    doanotherthing();
}
else
                    return;
unlock(&lk);
```

Buenas prácticas: cada return con su unlock

Mejor:

```
lock(&lk);
if (bla) {
                    dosomething();
                    doanotherthing();
}
else {
                    unlock(&lk);
                    return;
}
unlock(&lk);
```

Buenas prácticas: cada return con su unlock

Todavía mejor (no siempre se puede):

```
lock(&lk);
if (bla) {
                    dosomething();
                    doanotherthing();
}
unlock(&lk);
return;
```

Contention/Contienda

- Cuando dos flujos de ejecución compiten por entrar a la vez en la región crítica, uno espera.
- Si muchos intentan entrar $\rightarrow$ **contienda** elevada $\rightarrow$ muchos esperan.
- Eso es malo en general.
- Siempre debemos mantener las regiones críticas lo más pequeñas que se pueda.

Primitiva: lock/cerrojo

En general, hay dos tipos de cerrojos:

- **Spin locks**. El flujo que se bloquea en el cerrojo hace **espera activa**: el flujo consume tiempo de CPU iterando hasta poder coger el cerrojo. Sólo se deben usar cuando hay **poca contienda** y la región crítica es pequeña. No son justos.
- **Queue locks** (también conocidos como **mutexes**). El flujo que se bloquea en el cerrojo suelta el procesador y pasa a no estar listo para ejecutar. Cuando pueda coger el cierre, volverá a estar listo para ejecutar y el planificador le dará CPU cuando toque. Suelen ser FIFO (depende de la implementación).

Ejemplo de implementación de spin locks con TAS

```
void
lock(Lock *lk)
{
        int i;


        if(!_tas(&lk->val))             /* once fast */
                return;
        for(i=0; i<1000; i++){          /* a thousand times pretty fast */
                if(!_tas(&lk->val))
                        return;
                sleep(0);
        }
        for(i=0; i<1000; i++){          /* now nice and slow */
                if(!_tas(&lk->val))
                        return;
                sleep(100);
        }
        while(_tas(&lk->val))           /* take your time */
                sleep(1000);
}
```

Ejemplo de implementación de spin locks con TAS

```
void
unlock(Lock *lk)
{
        /* on AMD64, this is atomic */
        lk->val = 0;
}
```

Primitiva: cierre de lectores/escritores

Hay un tipo de cerrojo especial llamado lock de lectores/escritores. Tiene dos tipos de operaciones para coger el cerrojo:

- Los que van a leer pueden hacerlo concurrentemente (N lectores). Cogen el cierre en modo lectura.
- El que viene a escribir necesita estar solo (1 escritor). Coge el cierre en modo escritura.
- El nivel de concurrencia es mayor cuando hay muchas lecturas y pocas escrituras (caso común).

Otras primitivas

Además de los cerrojos, existen muchos otros mecanismos de sincronización con distinta semántica:

- Futex
- WaitGroup
- Semáforos
- Barreras
- Rendezvous
- Variables condición y monitores
- Canales
- ...

Si vamos a hacer programas concurrentes, tenemos que estudiar detenidamente los mecanismos ofrece el sistema en el que trabajamos y programar con **mucho cuidado**.

clone()

```
int clone(int (*fn)(void *), void *child_stack,
          int flags,  void *arg, ... );
```

Tiene muchas flags para compartir distintos recursos en padre e hijo:

- `CLONE_VM`: padre e hijo comparten memoria. Se comparte toda la memoria. Los mmap/munmap afectan a los dos.
- `SIGCHLD` para poder esperar por el hijo.
- La pila nunca se comparte (evidentemente). A `clone` se le pasa la pila para el hijo.
- Las pilas de los procesos NO están en distintos espacios de memoria
- Usar `clone` directamente puede ser complejo.

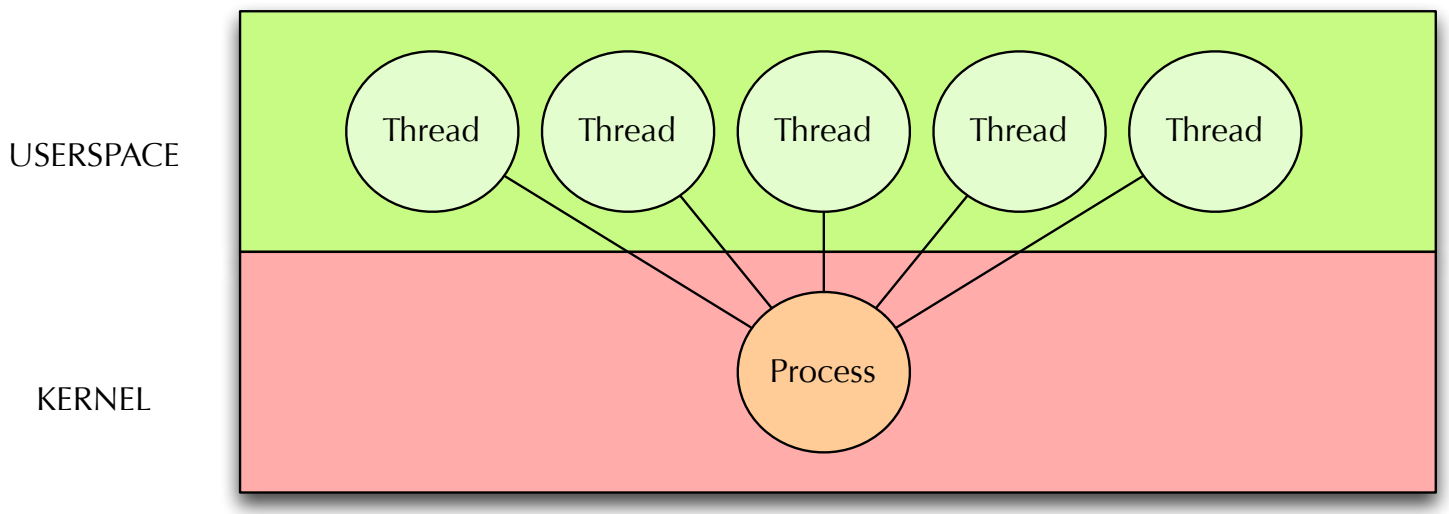
Procesos

Threads: su contexto es básicamente el estado de la CPU.

- $\downarrow$ transparencia de concurrencia $\Rightarrow \uparrow$ complejidad.
- Dos tipos de threads:
  - Threads de biblioteca (usuario).
  - Threads de kernel.
- Son expulsivos (preemptive) cuando el sistema se encarga de cambiar el contexto cuando toque (planificación).
- Son no expulsivos (non-preemptive), o colaborativos, cuando ellos deciden cuándo dejan la CPU para que pueda entrar otro.

Threads de biblioteca

- Modelo N-1.
- Cada thread tiene su propio contador de programa y pila.
- El contexto del thread se gestiona área de usuario. El OS no sabe nada de los threads.

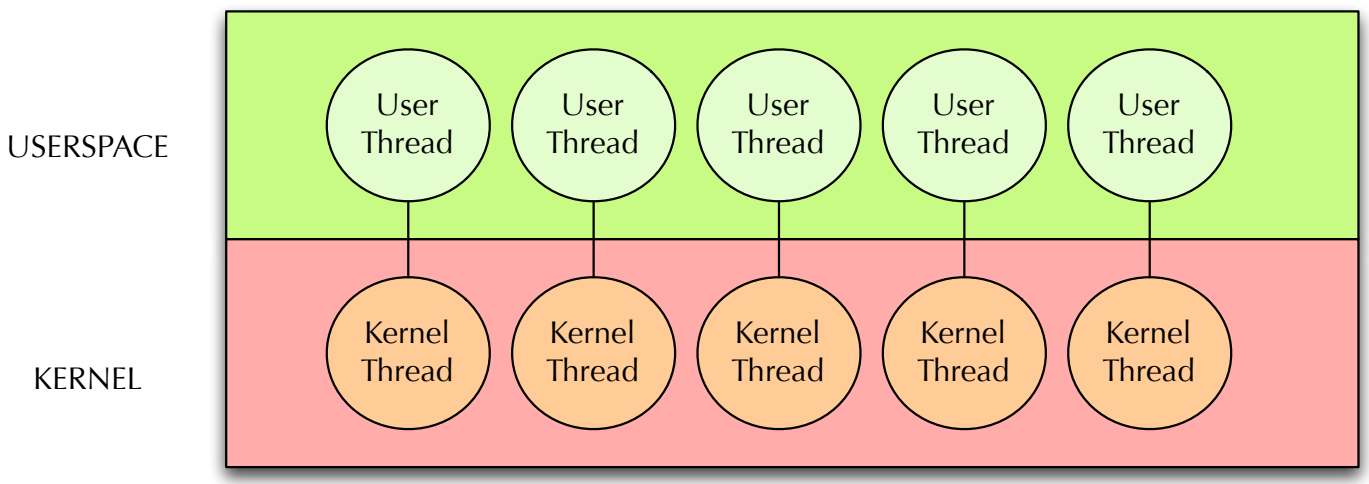


Threads de biblioteca

- Es barato crear, destruir y conmutar threads.
- Los threads no pueden ejecutar en paralelo en un multiprocesador.
- Si se bloquea el proceso se bloquean todos los threads. P. ej. I/O.

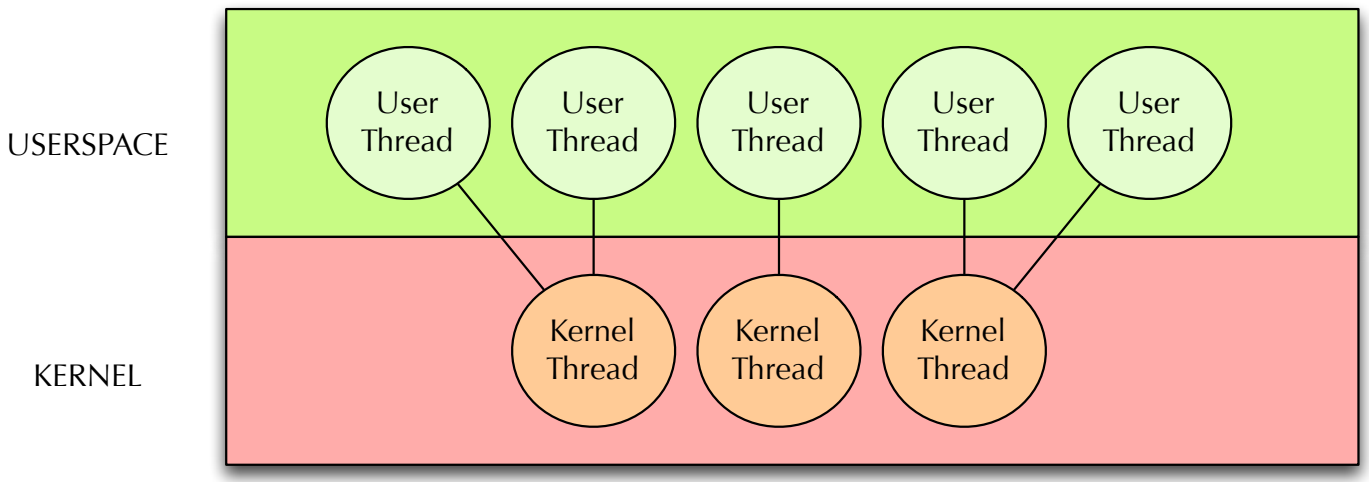
Threads de kernel

- Modelo 1-1.
- *Kernel-level threading*.
- Los threads son en realidad procesos que comparten memoria (y otras cosas): *procesos ligeros (LWP)*.
- Crear, destruir y conmutar threads más caro: hay que entrar al kernel.
- P. ej. Linux 2.6 NPTL (Native Posix Thread Library).



Threads de kernel

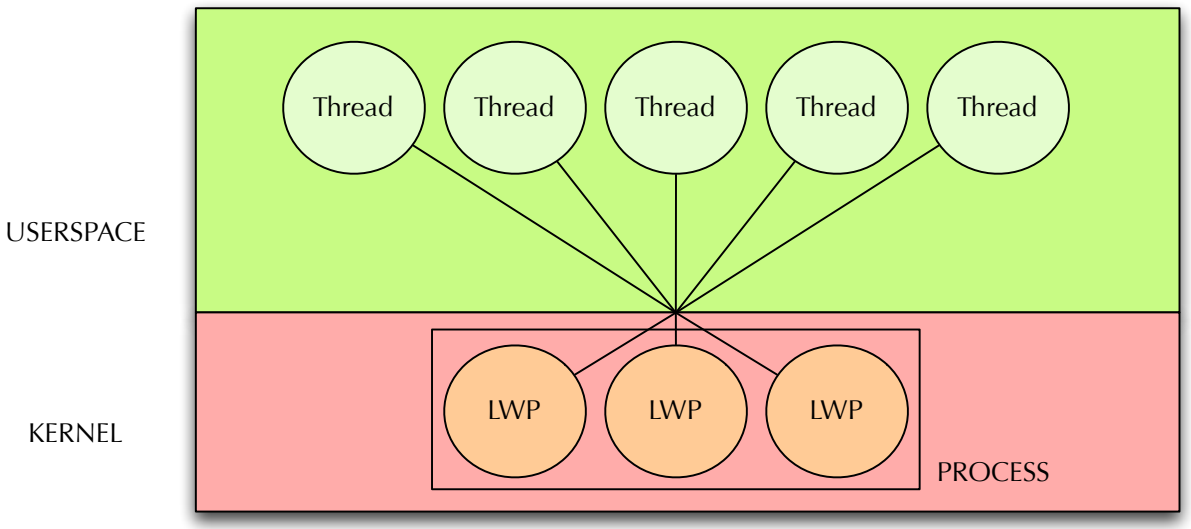
- Modelo N-M.
- Un proceso puede albergar uno o varios threads.
- Es un modelo híbrido que mezcla las dos aproximaciones anteriores.



Ejemplo M-N: SunOS LWP

- Un proceso se compone de un espacio de direcciones y un conjunto de LWPs, que el kernel planificará por separado.
- Un thread está totalmente representado en espacio de usuario.
- Un LWP es como una *CPU virtual* para un thread.
- Los LWP planifican los threads sin necesidad de entrar al kernel.
- Si un LWP se bloquea, otro LWP puede ejecutar el resto de threads del proceso. Cada LWP puede hacer llamadas al sistema, tener fallos de página y ejecutar en paralelo independientemente.

Ejemplo M-N: SunOS LWP



Pthreads

Pthreads

- Un thread (hilo) es la unidad mínima de utilización de CPU. Tiene un TID.
- Hay distintos modelos de threads.
- Por ahora, los podemos ver como procesos que comparten su memoria: TEXT, DATA, BSS.
- ¿Qué pasa con la pila?
- El estándar POSIX tiene su interfaz: pthreads. Cada sistema puede implementarla de una forma distinta.
  `man 7 pthreads`

Pthreads

- Los threads comparten las variables globales.
- Dogma: cada vez que toquemos un recurso compartido (p. ej. una variable compartida), nos tenemos que proteger.
- Si no hacemos eso, nos metemos en problemas.
- Si usas una biblioteca, asegúrate de que es *thread_safe*.
- En caso de duda, protege.

Pthreads

- `pthread_create()`: crea un thread, que comenzará ejecutando la función que se le pasa en su tercer parámetro. El primer parámetro es un puntero a la variable de tipo `pthread_t` que identificará al thread creado.
- `pthread_join()`: espera a que muera el thread indicado en su primer parámetro.

Debemos enlazar el programa con la biblioteca de threads:

```
gcc -c -Wall -Wshadow -g miprograma.c
gcc -o miprograma miprograma.o -lpthread
```

ver pthreads.c
ver pthreads-race.c

Cerrojos en pthreads

Tenemos:

- Spin locks: tipo de datos `pthread_spin_lock_t`.
- Mutex: tipo de datos `pthread_mutex_t`.

pthread_spin_lock_t

Operaciones básicas de `pthread_spin_lock_t`:

- `pthread_spin_init()`: inicializa el cerrojo. El segundo parámetro indica si se comparte con otros procesos (usar `PTHREAD_PROCESS_SHARED`).
- `pthread_spin_lock()`: coge el cerrojo.
- `pthread_spin_unlock()`: suelta el cerrojo.
- `pthread_spin_destroy()`: libera los recursos asociados al cerrojo.

ver pthreads-spin.c

pthread_mutex_t

Operaciones básicas de `pthread_mutex_t`:

- `pthread_mutex_init()`: inicializa el cerrojo. Si el segundo parámetro se usa para establecer ciertos atributos, si es NULL se ponen los atributos por omisión.
- `pthread_mutex_lock()`: coge el cerrojo.
- `pthread_mutex_unlock()`: suelta el cerrojo.
- `pthread_mutex_destroy()`: libera los recursos asociados al cerrojo.

ver pthreads-mutex.c

Sincronización con ficheros

Sincronización con ficheros

Cuando desde varios procesos se opera con el mismo fichero, tenemos condiciones de carrera. Ejemplos:

- Cuando varios procesos quieren crear el mismo fichero, hay una condición de carrera.
- Cuando un proceso escribe en un fichero mientras que otros leen/escriben del mismo fichero, tenemos una condición de carrera. En general, no se garantiza atomicidad en las escrituras.

ver writerace.c

Open

- El modo `O_CREAT|O_EXCL` hace que la llamada `open()` falle si existe el fichero que se quiere crear. En ese caso `errno` será `EEXIST`.
- Muchas aplicaciones usan esto como un TAS y la existencia de un fichero como un cerrojo (p. ej. Firefox para mantener una única instancia de la aplicación ejecutando).
- Hay que tener cuidado con la posición (offset). Si siempre queremos escribir al final: `O_APPEND`. Si múltiples procesos añaden concurrentemente, todos los bytes escritos se escribirán al final, pero puede que los bytes del mismo `write` no terminen contiguos.

Flock

- `flock`: sirve para usar un **cierre de lectores/escritores** sobre el fichero.
  Tiene tres operaciones:
  - `LOCK_EX`: echa el cierre de escritores.
  - `LOCK_SH`: echa un cierre de lectores.
  - `LOCK_UN`: soltar el cierre que tienes.
- Se puede especificar que no sea bloqueante con `|LOCK_NB`. En ese caso, si no puedes coger el cierre la operación da error (no se bloquea).

```
int flock(int fd, int operation);
```

ver filerace.c